# A New API for PCRE

| | |
|---|---|
| Revision: | 7 |
| Updated: | 18 April 2014 |
| Author: | Philip Hazel |

This document contains a proposal for a completely new API for PCRE. Changes from revision to revision of this document will be marked by vertical bars on the right, like this. Some familiarity with the old API is assumed because I haven't fully described what all the various functions and options actually do, especially when they are unchanged from the old API.

This new API does not have any user-visible C structures, except for *pcre2_callout_block*. Instead, function calls are used as the means of interacting with the library. This makes it easier to interface the library to languages other than C and C++ that cannot access C structure definitions or C macros.

## 1 Major changes to this revision

- Thinking about implementing some of this has caused me to revise my ideas about the 'context'. I now propose three different 'contexts' for data fields that are used at different times, and may be changed at different times. I also propose that simple PCRE2-using applications need not bother about contexts at all. They just pass NULL as context arguments. There are extensive changes as a result of this re-think.

## 2 Names and numbers

The new API will be introduced for release 9.0. In order to avoid confusion, especially when both APIs are simultaneously installed, the new API uses different names for functions, options, structures, and header files. All the new names begin with `pcre2` or `PCRE2`.

The libraries are called *libpcre2-8*, *libpcre2-posix*, *libpcre2-16*, and *libpcre2-32*, so that both old and new libraries may exist together. The names of the man pages all begin with `pcre2` for the same reason. The commands are renamed *pcre2grep* and *pcre2test*, though it might make sense to alias these to the old names if they do not already exist.

## 3 Handling different data widths

Every function comes in three different forms, for example:

```
pcre2_compile_8()
pcre2_compile_16()
pcre2_compile_32()
```

There are also three different sets of data types:

```
PCRE2_UCHAR8, PCRE2_UCHAR16, PCRE2_UCHAR32
PCRE2_SPTR8,  PCRE2_SPTR16,  PCRE2_SPTR32
```

The `UCHAR` types define unsigned code units of the appropriate widths. For example, `PCRE2_UCHAR16` is usually defined as 'unsigned short'. The `SPTR` types are constant pointers to the equivalent `UCHAR` types, that is, they are pointers to strings of unsigned code units.

Many applications use only one data width. For their convenience, macros are defined whose names are the generic forms such as *pcre2_compile()* and `PCRE2_UCHAR`. These macros use the value of the macro `PCRE2_DATA_WIDTH` to generate the appropriate width-specific function and macro names. `PCRE2_DATA_WIDTH` is not defined by default.

Applications that use more than one data width are advised not to define `PCRE2_DATA_WIDTH`, but instead to use the real function names, and any code that is to be included in an environment where the value of `PCRE2_DATA_WIDTH` is unknown should do likewise. (Unfortunately, it is not possible in C code to save and restore the value of a macro.)

In the rest of this document, functions and data types are described using their generic names, without the 8, 16, or 32 suffix. This can also be done in the user documentation for the new API.

## 4 Data blocks and multithreading

In a multithreaded application it is important to keep thread-specific data separate from data that can be shared between threads. The library code itself is thread-safe: it contains no static or global variables. The API is designed to be fairly simple for non-threaded applications while at the same time ensuring that multithreaded applications can use it.

There are several different blocks of data that are used to pass information between the application and the PCRE libraries.

- A pointer to the compiled form of a pattern is returned to the user when *pcre2_compile()* is successful. The data in the compiled pattern is fixed, and does not change when the pattern is matched. Therefore, it is thread-safe, that is, the same compiled pattern can be used by more than one thread simultaneously. An application can compile all its patterns at the start, before forking off multiple threads that use them.
- The new API introduces the idea of *contexts* in which PCRE functions are called. A context is nothing more than a collection of parameters that control the way PCRE operates. Grouping a number of parameters together in a context is a convenient way of passing them to a PCRE function without using lots of arguments. The parameters that are stored in contexts are in some sense 'advanced features' of the API. Many straightforward applications will not need to use contexts.

  In a multithreaded application, if the parameters in a context are values that are never changed, the same context can be used by all the threads. However, if any thread needs to change any value in a context, it must make its own thread-specific copy.
- The matching functions need a block of memory for working space and for storing the results of a match. This includes details of what was matched, as well as additional information such as the name of a `(*MARK)` setting. Each thread must provide its own version of this memory.

## 5 A simple example

In the old API, a straightforward use of PCRE, including studying the compiled pattern, looks like this:

```
pcre *re;
pcre_extra *extra;
const char *error;
int ovector[30];
int erroffset;
int rc;
re = pcre_compile("pattern", 0, &error, &erroffset, NULL);
if (re == NULL)
  {
  /* Handle error */
  }
extra = pcre_study(re, 0, &error);
if (error != NULL)
  {
  /* Handle error */
  }
rc = pcre_match(re, extra, "subject", 7, 0, 0, ovector, 30);
if (rc < 0)
  {
  /* Handle error */
  }
/* Use ovector to get matched strings */
```

```
pcre_free(re);
pcre_free_study(extra);
```

In the new API, error handling is different in order to accommodate 16- and 32-bit error messages, but studying is automatic. This simple example makes no use of contexts.

```
#define PCRE2_DATA_WIDTH 8   /* or 16 or 32 */
pcre2_code *re;
pcre2_match_data *match_data;
size_t &erroroffset;
size_t *ovector;
int errorcode;
int rc;
re = pcre2_compile("pattern", -1, 0, &errorcode,
  &erroffset, NULL);
if (re == NULL)
  {
  PCRE_UCHAR buffer[120];
  (void)pcre2_get_error_message(errorcode, buffer, 120);
  /* Handle error */
  }
match_data = pcre2_match_data_create(20, NULL);
rc = pcre2_match(re, "subject", -1, 0, 0, match_data, NULL);
if (rc < 0)
  {
  /* Handle error */
  }
ovector = pcre2_get_ovector_pointer(match_data);
/* Use ovector to get matched strings */
pcre2_match_data_free(match_data);
pcre2_code_free(re);
```

## 6 Managing contexts

There are three different types of context: a general context that is relevant for many PCRE operations, a compile-time context, and a match-time context. An application only needs to use them if it requires certain ‘advanced’ features such as custom memory management or non-standard character tables. The fields in a context are set via function interfaces.

### 6.1 Custom memory management

By default, PCRE uses the normal *malloc()* and *free()* functions for memory management. However, custom memory managers can be used by setting up a general context with pointers to external functions, and passing this context to the relevant functions. Whenever PCRE creates a data block of any kind using a custom memory manager, the block contains a pointer to the *free()* function in the general context that was used. When the time comes to free the block, this function is called.

### 6.2 The general context

At present, this context just contains pointers to (and data for) external memory management functions that are called from several places in the PCRE library. The context is named ‘general’ rather than specifically ‘memory’ because in future other fields may be added. A general context is created by:

```
pcre2_general_context_create(private_malloc, private_free,
  memory_data);
```

The two function pointers specify custom memory management functions, whose prototypes are:

```
void *private_malloc(size_t, void *);
void  private_free(void *, void *);
```

When code in PCRE calls these functions, the final argument is taken from the memory data field. Either of the first two arguments of the creation function may be NULL, in which case the system memory management functions are used. (This is not currently useful, as there are no other fields in a general context, but in future there might be.) The private malloc function is used (if supplied) to get memory for storing the context, and all three values are saved as part of the context.

## 6.3 The compile context

A compile context is created by:

```
  pcre2_compile_context_create(pcre2_general_context *);
```

If the argument is NULL, the memory for the context is obtained via *malloc()*. The fields are all initialized to default values, but can be changed by the following functions, all of which yield 1 for success or 0 if invalid data is given.

```
  pcre2_set_bsr_convention(compile_context, uint32_t bsr_code);
```

This specifies what characters the escape sequence `\R` matches. The allowed values are `PCRE2_BSR_UNICODE` (any Unicode newline sequence) or `PCRE2_BSR_ANYCRLF` (only CR, LF, or CRLF). The default is specified when PCRE is built.

```
  pcre2_set_character_tables(compile_context,
    unsigned char *tables);
```

This sets a pointer to custom character tables. The default is to use PCRE's inbuilt tables that were set up when it was built. A pointer to the tables is retained in a compiled pattern so that the same tables can be used at matching time.

```
  pcre2_set_newline_convention(compile_context,
    uint32_t newline_code);
```

This specifies which character codes are to be interpreted as newline. The second argument is one of `PCRE2_NEWLINE_CR`, `PCRE2_NEWLINE_LF`, `PCRE2_NEWLINE_CRLF`, `PCRE2_NEWLINE_ANY`, or `PCRE2_NEWLINE_ANYCRLF`. The default is specified when PCRE is built; it is normally the standard for the operating system.

```
  pcre2_set_parens_nest_limit(compile_context, uint32_t limit);
```

This sets the maximum depth of nested parentheses in a pattern. At compile time, each nesting causes a recursive function call. In environments with a limited system stack, too many of these may cause the stack to run out. The default is set when PCRE is built, with a default of 250.

```
  pcre2_compile_set_recursion_guard(compile_context,
    guard_function);
```

In some special environments the available system stack is very limited, and the use of recursive function calls at compile time may cause the stack to run out. The static limit provided by *pcre2_set_parens_nest_limit()* is often sufficient to deal with this issue. However, in some cases a dynamic check is required, and this function makes that possible. The guard function argument must be NULL to unset the feature, or a pointer to a callout function:

```
  int guard_function(int depth);
```

The argument contains the current recursion depth. If the function returns zero, compilation continues. Otherwise an error is generated. The external function can, of course, do any checks that it likes. It is not limited to checking the stack, though that is expected to be the most common use.

## 6.4 The match context

A match context is created by:

```
  pcre2_match_context_create(pcre2_general_context *);
```

If the argument is NULL, the memory for the context is obtained via *malloc()*. The fields are all initialized to default values, but can be changed by the following functions, all of which yield 1 for success or 0 if invalid data is given.

```
  pcre2_set_match_limit(match_context, uint32_t limit);
  pcre2_set_recursion_limit(match_context, uint32_t limit);
```

These values limit the resources used by a matching function (formerly passed in a *pcre_extra* structure). The default values are specified when PCRE is built, and are normally quite large.

```
  pcre2_set_callout(match_context, user_callout_function,
    user_data);
```

This records a user callout function. The prototype for the callout function is unchanged:

```
  int user_callout(pcre2_callout_block *, void *);
```

The callout block itself is also unchanged. Setting the function to `NULL`, which is the default, disables callouts.

```
  pcre2_set_recursion_memory_management(match_context,
    private_recursion_malloc, private_recursion_free);
```

This sets the alternate memory management functions that are used when PCRE is compiled to use the heap instead of the system stack for recursive function calls during interpretive matching. The memory blocks that are used for this purpose are all the same size, and are requested and freed in last-out-first-in order. A private memory manager could implement this kind of usage more efficiently than the general case. When PCRE is compiled to use the system stack for recursion, these additional memory management functions are never called.

### 6.5 Copying a context

An exact copy of a context can be made by:

```
  pcre2_general_context *new = pcre2_general_context_copy(old);
  pcre2_compile_context *new = pcre2_compile_context_copy(old);
  pcre2_match_context *new = pcre2_match_context_copy(old);
```

The memory for the new context is obtained using the same *malloc()* function as the old context.

### 6.6 Freeing a context

When a context is no longer needed, its memory can be freed by:

```
  pcre2_general_context_free(context);
  pcre2_compile_context_free(context);
  pcre2_match_context_free(context);
```

### 6.7 Reading parameter fields in a context

A previous version of this document specified many functions for extracting all the data from the (then) single context. I am not sure how useful such functions will be. For the moment, therefore, I haven’t specified any ‘get’ functions, but if they are needed, they can easily be added.

## 7 Compiling a pattern

A pattern is compiled by calling the following function:

```
  pcre2_code *pcre2_compile(
    PCRE2_SPTR             pattern,
    int                    length,
    uint32_t               options,
    int                   *error_code,
    size_t                *error_offset,
    pcre2_compile_context *compile_context);
```

If compile_context is NULL, the memory for the compiled pattern is obtained by calling the system *malloc()* function. Otherwise, this memory is obtained from the same function that was used to get the memory for the compile context.

The pattern is specified as a pointer and a length (number of code units). This is a change from the old API, where it was always a zero-terminated string. However, if the length is specified as a negative number, a zero-terminated string is assumed. Either or both of the contexts can be specified as NULL, which is expected to be the most common case. The following option bits are available:

| | |
|---|---|
| `PCRE2_ALLOW_EMPTY_CLASS` | allow [] as an empty class |
| `PCRE2_ALT_BSUX` | alternate behaviour for \U, \u, and \x |
| `PCRE2_ANCHORED` | pattern is anchored |
| `PCRE2_AUTO_CALLOUT` | generate auto callouts |
| `PCRE2_CASELESS` | assume caseless at start |
| `PCRE2_DOLLAR_ENDONLY` | $ matches only at the end |
| `PCRE2_DOTALL` | dot matches all characters |
| `PCRE2_DUPNAMES` | allow duplicate named subpatterns |
| `PCRE2_EXTENDED` | ignore white space in pattern |
| `PCRE2_FIRSTLINE` | must match before first newline |
| `PCRE2_MATCH_UNSET_BACKREF` | matches an empty string instead of error |
| `PCRE2_MULTILINE` | ^ and $ may match in mid-subject |
| `PCRE2_NEVER_UCP` | forbid `(*UCP)` in patterns |
| `PCRE2_NEVER_UTF` | forbid `(*UTF)` in patterns |
| `PCRE2_NO_AUTO_CAPTURE` | parentheses do not capture by default |
| `PCRE2_NO_AUTO_POSSESS` | disable auto-possessification |
| `PCRE2_NO_START_OPTIMIZE` | disable start-of-match optimization |
| `PCRE2_NO_UTF_CHECK` | disable pattern UTF validity check |
| `PCRE2_UCP` | use Unicode Properties for `\d` etc. |
| `PCRE2_UNGREEDY` | invert greediness |
| `PCRE2_UTF` | patterns and subjects are coded in UTF |

The `PCRE_EXTRA` option, which caused unknown escape sequences to give an error, like Perl's `-w` option, is now assumed always to be on.

The name `PCRE2_JAVASCRIPT_COMPAT` is not used. In the old API the use of the word JAVASCRIPT caused confusion because some people thought it gave full JavaScript compatibility. In fact, it caused only five changes to the way certain items were handled. One of those five (treating an isolated closing square bracket as an error instead of data) is no longer an issue, because JavaScript (aka ECMAScript) has changed. The other four are now supported by `PCRE2_ALLOW_EMPTY_CLASS`, `PCRE2_ALT_BSUX`, and `PCRE2_MATCH_UNSET_BACKREF`.

When successful, *pcre2_compile()* returns a pointer to an opaque structure that contains the compiled pattern and supporting information. This data is read-only, that is, it is never changed during pattern matching. Therefore, compiled patterns may be safely shared between threads.

If there is a compilation error, the function returns `NULL`. A positive error code and the offset in the pattern where the error occurred are placed in the variables pointed to by *error_code* and *error_offset*, respectively. The code can be translated into a textual error message by this function:

```
  int pcre2_get_error_message(int error_code, PCRE2_UCHAR *buffer,
    size_t buffer_size);
```

This copies a zero-terminated error message into the supplied buffer, whose code units are of the appropriate width (8, 16, or 32). It returns 1 if all is well, 0 if the buffer is too small. The old API always returned 8-bit error messages. If a 16- or 32-bit application wants an 8-bit error message, it can still obtain it by explicitly calling *pcre_get_error_message_8()*, but that does mean it will have to link with the 8-bit library as well as with the 16- or 32-bit one.

When a compiled pattern is no longer needed, it can be freed by:

```
pcre2_code_free(code);
```

## 8 Explicit studying is abolished

I introduced a separate *pcre_study()* function when PCRE was first implemented because I wasn't sure how much resource this would take. It turns out to be not very much (and processors are getting faster and faster). Therefore, in the new API, studying happens automatically and you don't have to worry about it. JIT compiling, on the other hand, is still expensive, so must remain optional.

## 9 JIT compiling

A compiled pattern may be further processed by the JIT compiler for faster matching:

```
int pcre2_jit_compile(pcre2_code *, jit_options);
```

The options must be at least one of the following:

| | |
|---|---|
| `PCRE2_JIT` | compile for non-partial JIT matching |
| `PCRE2_JIT_PARTIAL_SOFT` | compile for soft partial JIT matching |
| `PCRE2_JIT_PARTIAL_HARD` | compile for hard partial JIT matching |

The function yields 1 if JIT compilation was successful, zero otherwise.

## 10 Matching functions

The first releases of PCRE returned only the contents of *ovector*. When more information was needed, I found ways of passing it back in a compatible manner (e.g. in the *pcre_extra* block). It is time to tidy this up. I propose a new opaque structure called *pcre2_match_data*, which contains space for remembering the results of a match, and also contains working space for the matching function.

The primary data is in the *ovector*. This is now a vector of *size_t* instead of *int*. There is a special value, `PCRE_OVECTOR_UNSET`, probably defined as `(~(size_t)0)`, that is used for unset fields.

There are several different ways this might be set up. In a previous version of this document I laid out three different options. A small amount of feedback suggested that having the output vector managed just like the other data in the block was preferred. Therefore, the proposal is now as follows:

```
pcre2_match_data *match_data =
  pcre2_match_data_create(20, general_context);
```

This creates a *pcre2_match_data* block with room to store the offsets for 20 matched strings. The second argument can be NULL if custom memory management is not in use. An alternative form of this function is:

```
pcre2_match_data *match_data =
  pcre2_match_data_create_from_pattern(pcre2_code *);
```

This function obtains a size for the ovector from the number of capturing subpatterns, and it uses whatever memory management was used for the compiled code. When it is no longer needed, a match data block is freed by calling *pcre2_match_data_free()*. Access to the output vector is obtained by calling *pcre2_get_ovector_pointer()*, as described below, or by using one of the string-extracting convenience functions.

There was one request that it be possible to have the match data block on the stack. With a variable-sized block, as just defined, this cannot be done. Even if the block size were fixed (with a separate output vector) it would not be possible because, since the structure is opaque, the size is not known to the application.

## 11 Perl-compatible matching

The following function matches in a Perl-compatible manner:

```
int pcre2_match(
  const pcre2_code    *code,
  PCRE2_SPTR           subject,
  int                  length,
  size_t               startoffset,
  uint32_t             options,
  pcre2_match_data    *match_data,
  pcre2_match_context *match_context);
```

If any memory management is needed during matching, the functions associated with the match context are used, but if that argument is NULL, the functions associated with the match data are used.

A negative length means ‘zero-terminated string’. The match context may be NULL if none of its fields are required. The following option bits are available:

| | |
|---|---|
| `PCRE2_ANCHORED` | pattern is anchored |
| `PCRE2_NOTBOL` | subject is not the beginning of a line |
| `PCRE2_NOTEOL` | subject is not the end of a line |
| `PCRE2_NOTEMPTY` | must not match an empty string |
| `PCRE2_NOTEMPTY_ATSTART` | not empty at start of subject |
| `PCRE2_NO_START_OPTIMIZE` | disable start-of-match optimization |
| `PCRE2_NO_UTF_CHECK` | disable subject UTF validity check |
| `PCRE2_PARTIAL_SOFT` | soft partial match |
| `PCRE2_PARTIAL_HARD` | hard partial match |

The return codes are unchanged from the old API: zero or positive for a complete match, negative for error or a partial match. Information about the match is remembered in the match data block, including pointers to the compiled pattern and the subject string. You should not free the memory for these items until after any calls to extract information have taken place.

### 11.1 Matched strings

After a successful call to *pcre2_match()*, the offsets of the matched string and any captured substrings are saved in a vector within the match block. Individual strings can be copied into specified memory, or into newly obtained memory, using the string extraction functions described in the next section. Alternatively, the address of the output vector can be obtained by calling:

```
  size_t *pcre2_get_ovector_pointer(pcre2_match_data *);
```

The elements in the vector are used in pairs, as before. Note, however, that the value for an unset capturing group is `PCRE_OVECTOR_UNSET` instead of a negative number, because the vector is now of type *size_t*, not *int*.

The return code from *pcre2_match()* specifies how many strings have been captured. When there are more strings than vector slots, zero is returned. For the convenient handling of this case, the function

```
  size_t pcre2_get_ovector_count(pcre2_match_data *);
```

returns the number of pairs of offsets.

## 12 String extraction functions

As the memory management, output vector, and the details of the most recent match are remembered in the match data, there is no need to pass them to the string extraction functions. I have renamed the functions to try to make them consistent in style with the other functions. Apart from changes to the names and variable types, the following functions are otherwise almost the same as the old API:

```
  int pcre2_substring_copy_bynumber(
    pcre2_match_data *match_data,
    int stringnumber,
    PCRE2_UCHAR *buffer,
    size_t buffsize);

  int pcre2_substring_copy_byname(
    pcre2_match_data *match_data,
    PCRE2_SPTR name,
    PCRE2_UCHAR *buffer,
    size_t buffsize);

  int pcre2_substring_get_bynumber(
    pcre2_match_data *match_data,
    int stringnumber,
    PCRE2_UCHAR **);

  int pcre2_substring_get_byname(
    pcre2_match_data *match_data,
    PCRE2_SPTR name,
    PCRE2_UCHAR **);

  void pcre2_substring_free(
    PCRE2_SPTR string);

  int pcre2_substring_number_from_name(
    const pcre2_code *code,
    PCRE2_SPTR name);

  int pcre2_substring_nametable_scan(
    const pcre2_code *code,
    PCRE2_SPTR name,
    PCRE2_UCHAR **first,
    PCRE2_UCHAR **last);
```

In the old API, the function *pcre_get_substring_list()* is inadequate, because it cannot process substrings containing binary zeroes. An extra argument is added so that the function can return a list of lengths as well as as a list of pointers to the substrings. All this data is placed in a contiguous memory block that can be freed by a single function call.

```
  int pcre2_substring_list_get(
    pcre2_match_data *match_data,
    PCRE2_UCHAR ***,
    size_t **);

  void pcre2_substring_list_free(
    PCRE2_SPTR *list);
```

Two additional functions are provided:

```
  int pcre2_substring_length_bynumber(
    pcre2_match_data *match_data,
    int stringnumber);

  int pcre2_substring_length_byname(
    pcre2_match_data *match_data,
    PCRE2_SPTR name);
```

These return the length of a matched substring, specified by number or by name, respectively. If there is no such substring, the negative value PCRE_ERROR_NOSUBSTRING is returned. If the substring did not participate in the match, zero is returned.

### 12.1 Additional match information

As well as the offsets that define a successful match, other data from the most recent match is remembered, whether it succeeded or failed. This can be extracted from the match data using the following functions:

```
  PCRE2_SPTR pcre2_get_mark(pcre2_match_data *);
```

If the match found a `(*MARK)` name to pass back, a pointer to it is returned. Otherwise the function returns NULL. The name is a zero-terminated string within the compiled pattern (as before).

```
size_t pcre2_get_startchar(pcre2_match_data *);
```

This function returns the offset of the character where the final matching process began. For an anchored pattern, the value is always *startoffset*. This offset is different to the starting offset of a matched string if `\K` was encountered.

```
size_t pcre2_get_leftchar(pcre2_match_data *);
size_t pcre2_get_rightchar(pcre2_match_data *);
```

These functions return the offsets of the leftmost and one more than the rightmost characters that were inspected during the final match attempt. Lookbehinds and lookaheads can make these offsets less than or greater than the offsets of a matched string, respectively. For example, when the pattern

```
(?<=abc)def(?=ghi)
```

is matched against the string "`abcdefghi`" the offsets of the matched string are 3 and 6, corresponding to "`def`", whereas the leftmost and rightmost offsets are 0 and 9.

## 12.2 Errors while matching

An error message can be obtained for any error code using the same function as for *pcre2_compile()*:

```
int pcre2_get_error_message(int error_code, PCRE2_UCHAR *buffer,
  size_t buffer_size);
```

The offset in the subject where the error occurred can be obtained by:

```
size_t pcre2_get_error_offset(pcre2_match_data *);
```

When the error is `PCRE2_ERROR_BADUTF8` or `PCRE2_ERROR_SHORTUTF8`, another function can be called to obtain a detailed reason code:

```
int pcre2_get_error_reason(pcre2_match_data *);
```

This yields values such as `PCRE2_UTF8_ERR1` (truncated UTF-8 character). The use of these functions replaces the previous rather untidy scheme of putting values into the output vector.

## 12.3 Change to partial matching

In the current API, after a partial match, the first three values in *ovector* are the leftmost character, the end of the partial match (the end of the subject), and the start match offset. (This is because there have been various changes and additions over the years.) In the new API, only two values are set in the output vector, and they are those of the partially matched string, giving consistency with a complete match. The other offsets are now available for all matches using the functions just described.

## 12.4 Obtaining the frame size

In the current API, a call to *pcre_match()* with NULL arguments is a convention for obtaining the size of the stack or heap frame (depending on how PCRE was compiled) that is used for recursive calls of the matching function. This facility is now provided by a separate function:

```
int size = pcre2_get_match_frame_size();
```

## 13 DFA matching

The DFA matching function needs workspace. A vector of at least 20 *int*s is recommended; more is needed for patterns that have a lot of potential matches. I have used additional arguments, as in the current API, but an alternative would be to have a separate *pcre2_dfa_match_data* block, in which case the rules for handling the workspace should be the same as for handling the output vector.

```
int pcre2_dfa_match(
  const pcre2_code     *code,
  PCRE2_SPTR           subject,
  int                  length,
  size_t               startoffset,
  uint32_t             options,
  pcre2_match_data    *match_data,
  pcre2_match_context *match_context,
  int                 *workspace,
  size_t               wscount);
```

The following option bits are available in addition to those of *pcre2_match()*:

| | |
|---|---|
| `PCRE2_DFA_RESTART` | restart after a partial match |
| `PCRE2_DFA_SHORTEST` | find only the shortest match |

## 14 JIT matching

The functions for detailed JIT matching are adjusted for the new API in fairly obvious ways:

```
pcre2_jit_stack *pcre2_jit_stack_alloc(pcre2_general_context *,
  size_t, size_t);
void pcre2_jit_stack_free(jit_stack);
void pcre2_jit_stack_assign(const pcre2_code *,
  pcre2_jit_callback, void *);
void pcre2_jit_free_unused_memory(pcre2_general_context *);
int pcre2_jit_match(const pcre2_code *, PCRE2_SPTR,
  int, size_t, uint32_t, pcre2_match_data *, pcre2_jit_stack *);
```

The abolition of *pcre_extra* means that a *pcre2_code* pointer is passed instead. The arguments for *pcre2_jit_match()* are now the same as the new *pcre2_match(),* plus a JIT stack pointer.

## 15 Pattern information

The function for obtaining information about a compiled pattern is now:

```
int pcre2_pattern_info(const pcre2_code *, uint32_t, void *);
```

Many of the information items are unchanged, but I have removed those that are obsolete or deprecated, and done some renaming. This is the new list, with further comment below on those that are changed:

| | |
|---|---|
| `PCRE2_INFO_BACKREFMAX` | highest backreference |
| `PCRE2_INFO_CAPTURECOUNT` | number of capturing subpatterns |
| `PCRE2_INFO_COMPILE_OPTIONS` | options set for compile |
| `PCRE2_INFO_FIRSTCODEUNIT` | value for first code unit |
| `PCRE2_INFO_FIRSTCODETYPE` | type of first code information |
| `PCRE2_INFO_FIRSTTABLE` | table of first data values |
| `PCRE2_INFO_HASCRORLF` | has explicit CR or LF |
| `PCRE2_INFO_JCHANGED` | `(?J)` or `(?-J)` was used |
| `PCRE2_INFO_JIT` | successful JIT compilation |
| `PCRE2_INFO_JITSIZE` | JIT compiled code size |
| `PCRE2_INFO_LASTCODEUNIT` | value for last code unit |
| `PCRE2_INFO_LASTCODETYPE` | type of last code information |
| `PCRE2_INFO_MATCH_EMPTY` | can match an empty string |
| `PCRE2_INFO_MATCHLIMIT` | limit set within the pattern |
| `PCRE2_INFO_MAXLOOKBEHIND` | maximum lookbehind, in characters |
| `PCRE2_INFO_MINLENGTH` | minimum length, in characters |
| `PCRE2_INFO_NAMECOUNT` | number of name table entries |
| `PCRE2_INFO_NAMEENTRYSIZE` | size of each entry |
| `PCRE2_INFO_NAMETABLE` | pointer to the name table |
| `PCRE2_INFO_PATTERN_OPTIONS` | options set within the pattern |

|   |   |
|---|---|
| `PCRE2_INFO_RECURSIONLIMIT` | limit set within the pattern |
| `PCRE2_INFO_SIZE` | size of compiled pattern (bytes) |

Options that are explicitly passed to *pcre2_compile()* and those that are deduced from the pattern, for example, by the use of `(?i)`, are saved separately in the updated code; hence the splitting of `PCRE_INFO_OPTIONS` into two new options.

`PCRE_INFO_STUDYSIZE` was only ever provided to make it possible to save and restore the separate study data, so it is no longer relevant.

`PCRE2_INFO_FIRSTCODETYPE` gives information about the first code unit in a non-anchored pattern. If there is a fixed first value, for example, the letter `c` from a pattern such as `(cat|cow|coyote)`, 1 is returned. In this situation, the value can be retrieved using `PCRE2_INFO_FIRSTCODEUNIT`.

If there is no fixed first value, and if either (a) the pattern was compiled with the `PCRE2_MULTILINE` option, and every branch starts with "^", or (b) every branch of the pattern starts with ".*" and `PCRE2_DOTALL` is not set (if it were set, the pattern would be anchored), 2 is returned, indicating that the pattern matches only at the start of a subject string or after any newline within the string. Otherwise 0 is returned. For anchored patterns, 0 is returned. In all these cases, `PCRE2_INFO_FIRSTCODEUNIT` returns 0.

`PCRE2_INFO_LASTCODETYPE` returns 1 if there is a rightmost literal code unit that must exist, other than at the start of the subject, for a match to be possible. Otherwise it returns 0. In situations where 1 is returned, `PCRE2_INFO_LASTCODEUNIT` can be used to retrieve the value. In other cases, it returns 0.

## 16 Reference counts

The old API contains a function called *pcre_refcount()* which can be used to maintain a reference count within a compiled pattern. This breaks the assumption that a compiled pattern is a read-only structure. Also, it is not atomic, and therefore not thread-safe.

I do not think that introducing thread-specific functions such as atomic updates into the API just for this case is a good idea because it complicates the code and the specification, and makes building PCRE difficult in environments that do not support threads. Though it has been in PCRE since release 6.0, I propose to abolish *pcre_refcount()*.

Applications that need to maintain reference counts should instead define their own structure, something like this:

```
struct my_code {
  pcre2_code *code;
  int refcount;
  ...whatever...
};
```

Then they can manipulate the reference count any way they like, and the *pcre2* structure remains read-only.

## 17 Character tables

I propose no change to the way PCRE handles character tables, so this function remains:

```
const unsigned char *pcre2_maketables(pcre2_general_context *);
```

Note, however, that the pointer to custom character tables is now held in the compile context.

## 18 Configuration information

There is no change to the function for obtaining configuration information:

```
int pcre2_config(int what, void *where);
```

The available information is unchanged. However, building PCRE is simplified so that including UTF always also includes Unicode property support (see below), so `PCRE_CONFIG_UCP` is removed.

## 19 PCRE version

For consistency with the rest of the API, *pcre_version()* is changed to:

```
  int pcre2_version(PCRE2_UCHAR *buffer, size_t size);
```

The version and date string is copied into the supplied buffer, with a terminating zero. This allows the different libraries to return the version information in code units of the appropriate width. The function returns the length of the string (excluding the terminating zero) on success, or the negative error PCRE_ERROR_BADLENGTH if the buffer is too small. 16- and 32-bit applications that nevertheless want an 8-bit version string can obtain it by explicitly calling *pcre_version_8()*.

## 20 Byte-ordering functions

The prototypes for these functions are the obvious adaptions:

```
  int pcre2_pattern_to_host_byte_order(pcre2_code *);
  int pcre2_utf16_to_host_byte_order(PCRE2_UCHAR16 *, PCRE2_SPTR16,
    int, int *, int);
  int pcre2_utf32_to_host_byte_order(PCRE2_UCHAR32 *, PCRE2_SPTR32,
    int, int *, int);
```

## 21 Pre-compiled patterns

The facility for saving and restoring pre-compiled patterns is, I believe, used, so it should be preserved. The new code combines what was formerly separate study data into the main pattern structure, which makes things simpler, and the existing instructions for saving and restoring might continue to work in the new API. However, I think it would be better to provide explicit writing and reading functions; details yet to be worked out. They could incorporate the function of *pcre2_pattern_to_host_byte_order()*, which could then be abolished.

## 22 C++

The C++ wrapper supports only the 8-bit library and is currently not maintained. Unless a maintainer comes forward, I think it would be better to discard it. A new version should support 8-bit, 16-bit and 32-bit handling.

## 23 Substitution function

There have been requests for a substitution (find and replace) function. The existing C++ wrapper contains such a function, so maybe now is the time to provide one in the main library. Here is a possible specification:

```
  int pcre2_substitute(
    pcre2_code *code,              compiled pattern
    PCRE2_SPTR subject,            subject string
    int slength,                   length of subject string
    size_t startoffset,            offset to start search
    uint32_t options,              pcre2_match() options
    PCRE2_SPTR replacement,        replacement string
    int rlength,                   length of replacement string
    PCRE2_UCHAR *buffer,           where to put result string
    size_t blength,                length of buffer
    size_t *rlength);              where to return result length
```

The first six arguments are the same as the arguments for *pcre2_match()*.

**Question:** Should there be a *pcre2_match_data* argument? I have not specified this because the substitution function can get one for itself and free it when finished. It can find the number of captured substrings in order to set up an appropriate output vector. This uses more resources, but this is after all a convenience function. An application that is worried about performance would probably use its own code instead.

The allowed options, except for the partial matching options, are the same as for *pcre_exec()*. The replacement string is given as a pointer and a length so that binary strings can be processed. A negative length indicates a zero-terminated string. The string may contain substitution fragments in these forms:

|  |  |
|---|---|
| `$<number>` | e.g. `ab$1cd` |
| `${<number>}` | e.g. `12${3}34` |
| `$name` | e.g. `a $name b` |
| `${name}` | e.g. `a${name}b` |

The modified string is placed in *buffer,* whose length is *blength*. For the convenience of applications that are processing zero-terminated strings, a zero code unit is added at the end. The length of the modified string (excluding the terminating zero) is placed in the variable pointed to by *rlength*. The function returns the length of the initial copied substring plus the length of the expanded replacement string. This is the offset to ‘the rest of the string’.

If there is an error, a negative error code is returned. `PCRE2_ERROR_NOMATCH` is given for no match, and `PCRE2_ERROR_BADLENGTH` if the buffer is not large enough.

By default, this function does a single substitution on the first match that is found in the subject string. The value returned by the function is the offset in the modified string at which to start the next search. However, if the `PCRE2_GLOBAL_SUBSTITUTE` option is set, multiple substitutions occur, and the value returned is the same as the value that is placed in *rlength*.

## 24 Build-time changes

Originally, UTF support was implemented without UCP support, so when the latter was added later, it was made optional. Perhaps this is nowadays rather pointless; UTF should imply UCP.

## 25 The POSIX wrapper

There can, of course, be no change to the API for the POSIX wrapper. The revised functions for the new API will use PCRE contexts with default settings.

## 26 The pcretest program

I hacked up *pcretest* as a quick tester, and it has got more and more hackier as time has passed. Changing it is independent of a new library API, but if it is to be re-written, now is a good time to do it. The following are planned:

- Redesign the code to be more easily understood.
- Redesign the options syntax, both for patterns and subject strings.
- Invent a way of specifying pattern options to apply to all subsequent patterns until further notice.
- Allow comments in the input without treating them as patterns.

-oOo-